UPPSALA
UNIVERSITET

NATURAL COMPUTATION METHODS FOR MACHINE LEARNING

Project Report - Spring 2020

Reinforcement learning implementation in Unity using PPO algorithm

Group 07

Abhinav Singh, Praveen Swamy and Uvais Karni Mohideen Meera Sha

Aug 10,2020

Contents

1 Introduction

Reinforcement learning is an important machine learning method, an online learning technology where an action determined by the policy is rewarded with an appraisal. The intelligent agent improvises its action strategy to adapt to the environment based on the appropriate appraisal obtained from the actions determined. In recent years, a family of policy gradient methods for reinforcement learning was introduced which perform one gradient update per data sample and were relatively complicated to understand and implement. Therefore, a much simpler RL algorithm, which is Proximal Policy Optimization(PPO) [1] was invented, that strikes a perfect balance between the ease of implementation, sample efficiency, and ease of tuning [1].

In this project we solved a simple maze game wherein the agent was trained to find the shortest path towards the target with minimal collisions with the walls and the obstacles. Unity 3D[2] - a game development platform is used for creating a maze-like environment. ML-agents toolkit, an open-source Unity project is used, which enables the training of the intelligent agents using reinforcement learning or any machine learning methods through a simple-to-use Python API.

Then different features of the Unity were utilized to analyze and monitor the performance of the policy optimization e.g. multiple environment spaces were built, creating replicas of the same environment to monitor the speed of convergence while parallel training, played with the Curriculum learning[5], shaping the rewards[4] i.e. finding the right amount of reward that needs to be assigned for each state-action pair, tweaking the parameters for Ray-perception 3D sensor[2] for the agent and finally gathering the results and analyze the training statistics with the plots.

2 Methods

Before we start with the core idea some basic details to be aware of. In Reinforcement learning an agent, chooses or selects the action based on a policy. The policy gives the probability of possible action in a given state. An algorithm such as PPO is used to find the policy with the maximum expected return. There are two types of Stochastic policies that can be used Categorical and Gaussian. In our problem, we use Categorical policies because it supports discrete action space. The goal of RL is to select a policy which maximises the expected return. In order to maximise the policy returns, we need an optimization algorithm like policy gradient

2.1 Policy Gradient

The main idea behind the policy gradient is to increase the probability of actions that will lead to higher return and the opposite for the actions that will return lower returns which are achieved by perturbing the policy parameters in many different ways, measuring the performance and then moving in the direction of good performance. For each episode, the states, actions, and rewards encountered by the agent are saved in the memory. From the above values, a discounted future return is calculated for each episode. These discounted future returns are used as weights and actions the agent took as labels to perform backpropagation to update weights of the Neural Network.

2.1.1 Gradient Function

The gradient function is used to update the policy parameter, the mentioned function finds the gradient step to maximise the reward. The gradient for the policy is calculated by the given function.

$$\nabla_\theta J(\theta) \propto \sum_i (\sum_t \nabla_\theta \log \pi_\theta(a_t^i|s_t^i)(\sum_t r(s_t^i, a_t^i))) \tag{1}$$

$Where$,

a : action taken by the agent.

s : state of the agent.

θ : policy parameter.

π_θ : policy.

$\pi_\theta(a_t|s_t)$: probability to take action on a given state.

$r(s_t, a_t)$: reward for a given action at a given state.

2.1.2 Update Function

The update function takes the gradient step to update the previous policy parameter which in turn increases the reward attained by the previous policy.

$$\theta_{t+1} = \theta_t + \nabla_\theta J(\theta_t) \tag{2}$$

$Where$,

θ_t : old policy parameter.

θ_{t+1} : updated policy parameter.

$\nabla_\theta J(\theta_t)$: policy gradient.

In the end, we get policy that maximises the probability of taking actions with high expected future returns.

2.1.3 Issues of policy gradient

Vanilla Policy gradient suffers from high variance, low convergence and is not sampled efficiently, as previously stored states, actions, and rewards are discarded after every episode, but the updated weights are not affected(simply discard experience after every episode). The policy gradient does not work well with a dynamic environment or environment with large spaces because they will both have too many combinations.

2.1.4 Solution to above issue

Simply the above issues can be resolved by using batch-based updating, which results in faster convergence as it helps control the variance which was caused due to different episode length in the vanilla policy gradient. Enhanced version of the policy gradient such as Proximal Policy Optimization and Trust Regional Policy Optimization can be used to resolve the issue faced.

2.2 TRPO

TRPO is what PPO is based on, TRPO(Trust Region Policy Optimization) main aim is to take largest steps possible to maximise rewards without jeopardising the performance which is possible by the use of KL divergence constraint. The reason being that even small steps in parameter against the gradient will result in unrecoverable state or large disruption, simply a lot slower convergence.

2.2.1 KL-Divergence

The KL-Divergence(Kullback–Leibler divergence) measures how one probability distribution(old policy) is different from second probability distribution(updated policy), this constraint is used to limit the closeness of the two policy. It helps constraint the step size in the parameter space.

$$L(\theta_k, \theta) = E_{s,a\sim\pi_{\theta_k}}[\frac{\pi_\theta(a|s)}{\pi_{\theta k}(a|s)} A^{\pi_{\theta_k}}(s,a)] \tag{3}$$

$Where$,

$\frac{\pi_\theta(a|s)}{\pi_{\theta k}(a|s)}$: denotes the ratio of probability under the new and old policy.

π : is the policy.

$E_{s,a\sim\pi_{\theta_k}}$: denotes the empirical expectation.

$A^{\pi_{\theta_k}}(s,a)$: is the estimated advantage, which is the expected rewards minus a baseline like V(s) (A(s,a) = Q(s,a)-V(s,a)).

2.2.2 Issues of TRPO

TRPO attains maximum return through means of complex second order method KL-Divergence term. PPO uses the first order and other approximation to reach the same concluded results.

2.3 PPO

The motivation behind the invention of PPO relies on the same point as TRPO's, which is how can we take the largest possible step to optimize the policy using the current data without causing any damage to the current policy's performance. In case of TRPO the developers solved this issue by using KL divergence term. However, PPO is a family of first-order methods and uses a clipped surrogate objective function (described in equation 4) to keep new policies closer to the old policy.

PPO is an on-policy algorithm which is derived from the TRPO and can be used with the environments having both discrete and continuous action spaces. PPO being an on-policy type, doesn't store any past experiences but instead learns directly by interacting with the environment [1].

As described in the previous topics, Policy gradient methods are too sensitive with the choice of hyper-parameters [1] – too small step size leads to very slow progress and too large step size will indulge noise and leads to very poor performance. PPO updates policies using clipped surrogate objective function which constraints the policy change in a small range using a clip function.

$$L^{clip}(\theta) = E_t[min(r_t(\theta)A_t, clip((r_t(\theta), 1-\epsilon, 1+\epsilon)A_t))] \tag{4}$$

$$Where, r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta k}(a|s)}$$

$r_t\theta$: denotes the ratio of probability under the new and old policy.

π : is the policy.

θ : is the policy parameter.

E_t : denotes the empirical expectation over time-steps.

A_t : is the estimated advantage at time t, which is the expected rewards minus a baseline like V(s) (A(s,a) = Q(s,a)-V(s,a)).

ϵ : is a (small) hyper-parameter which roughly says how far away the new policy is allowed to go from the old.

clip : is a function used to clip the probability ratio r at 1-ϵ or 1+ϵ depending on whether the advantage is positive or negative.

Consider a single state-action pair (s,a) to understand a few cases for the advantage function - A.

1. **Advantage, A > 0 :** In this case (illustrated in figure 1), when the advantage for the state-action pair is positive then the objective increases if the action becomes more likely i.e. if $\pi_\theta(a|s)$ increases. The min function puts a limit on how much the objective function can increase. Once the objective value crosses 1+ϵ, then min function puts a limit on this value. Thus, the new policy does not diverge far away from the old policy.

2. **Advantage, A < 0 :** In this case (illustrated in figure 2), when the advantage is negative the action must be discouraged due to which the objective will decrease i.e. $\pi_\theta(a|s)$ decreases. But due to the clip function the objective will decrease to as little as 1-ϵ. Thus, the clip function prevents the agent from getting too greedy leading to a good approximation[1].

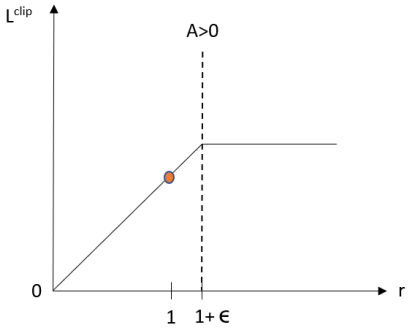

Figure 1: Advantage is positive

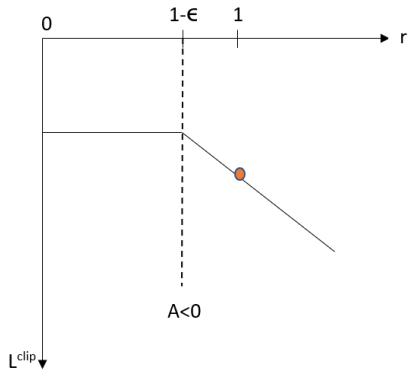

Figure 2: Advantage is negative

So far we have seen that how the clip function acts as a regularizer by restricting the policy, and the ϵ hyper-parameter makes sure that the new policy doesn't diverge too far from the old policy while still maintaining the performance of the objective function.

3 Implementation

1. **Tools used:** Unity 3D[2], ML-agents toolkit[2], Tensor board
2. **Programming language:** C#, Python
3. **Libraries:** TensorFlow

3.1 Procedure

A simple maze environment was built along with the agent and the target. ML-agents toolkit which is an API is used to train the agent using the pre-implemented PPO algorithm. Unity provides various features like ray-perception sensor, box colliders. In this maze game, box colliders play an important role in providing obstruction behaviors for the objects. The ray perception sensor is used to project n numbers of rays at specified angles and length, this allows us to detect the objects colliding within the proximity of the rays.

Once the environment is built and the above-mentioned features are setup, next step is to collect the observations of the agent from the environment which are inputs to the algorithm. This can be done using the C# script wherein we use the built-in functions. The observations for the agent will be agent's position, target's position, agent's velocity, distance between agent and target and finally the vector consisting the objects detected by the ray perception sensor. We also set the rewards for the possible state-action pair in the script.

The hyper-parameters for training the PPO algorithm can be altered if needed using the default yaml config file provided with the ML-agents toolkit. Once all the above steps are completed, we can commence with the training phase. During this training phase, the agent starts exploring the environment space. The policy gets updated only after T(max steps) number of time steps. The episode can end based on two conditions, when the agent reaches the target or when the maximum T steps are encountered. This T max steps should not be confused with the max_steps specified in the config file as the later specifies the maximum steps allowed for the entire training process.

Once the network is trained, we can use the trained model file to infer the results. Also, the training statistics can be visualized using the Tensor-board framework.

3.2 Insights

3.2.1 Reward shaping

Shaping the rewards [4] i.e. defining the right rewards accelerates the learning process and leads to faster convergence[4]. In this case, the agent was rewarded with a smaller negative value -0.001 for each action which makes the agent find the shortest path towards the target, then a higher negative value -0.1 for collisions with the walls and the maximum positive reward +10 for finding the target.

3.2.2 Curriculum learning

Curriculum learning[5] can be used in environments to accelerate the converging process by gradually increasing the difficulty level. In this case, the target's position was incrementally moved away from the agent after each episode. This way the agent was able to reach the target faster and avoid getting stuck in corners or dead ends of the maze. PPO being a simplified version, tweaking of hyperparameters was not a crucial factor to obtain good results. Therefore, the default values defined for the PPO helped in achieving the optimized training performance and reach the target.

4 Results

We were successfully able to implement PPO algorithm in Unity using ML-agents toolkit. Training the agent using curriculum learning is a better way as it allows the network to converge faster. In our case it was almost two times faster which can be seen in the plots below.

4.1 Cumulative reward vs Episodes without curriculum learning

During the initial episodes of training, the agent starts exploring the environment and learns to avoid hitting the walls since it gains a negative reward for each collision with the wall (from 0 to 10000 episodes). After 50,000 episodes the agent starts finding the target and then it finally converges to an optimal solution after 180,000 episodes.

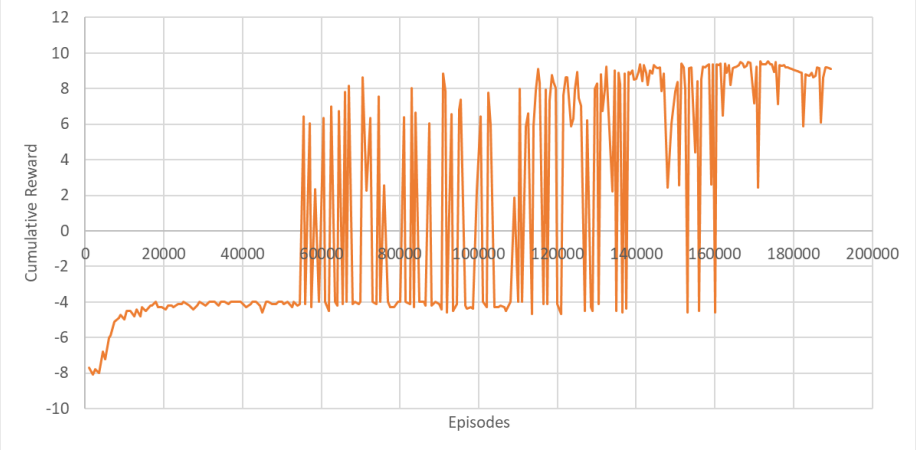

Figure 3: Cumulative reward vs Episodes without curriculum learning

4.2 Cumulative reward vs Episodes with curriculum learning

In our maze game to increase the difficulty level was to move the target further away from the agent. As the target is close to the agent it converges faster around 25,000th episode. After which the difficulty level is increased i.e. the target is moved more further away and thus there is a drop in the reward value around 30,000th episode. The same trend continues until around 90,000 episodes. After which it converges to the optimal solution.

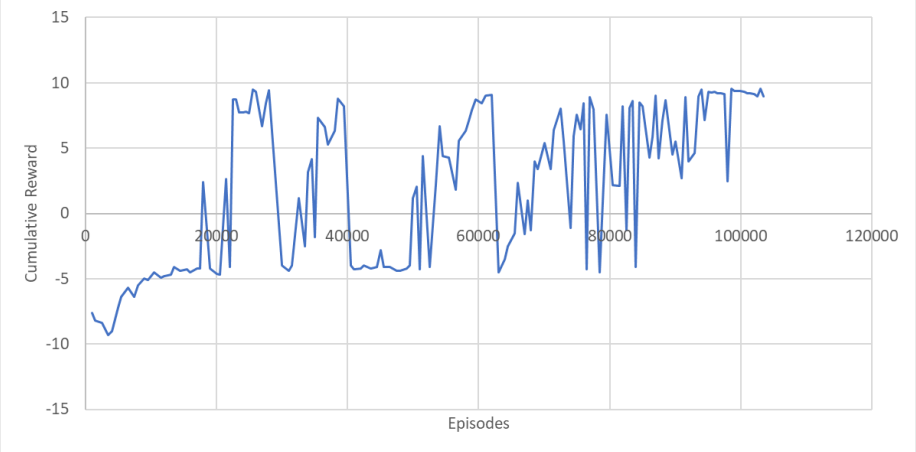

Figure 4: Cumulative reward vs Episodes with curriculum learning

Three maze environments developed are as shown in figures 5,6 and 7.

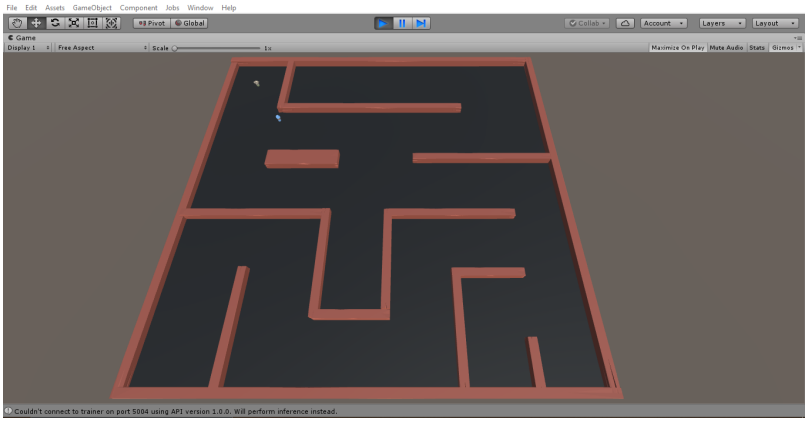

Figure 5: Maze 1

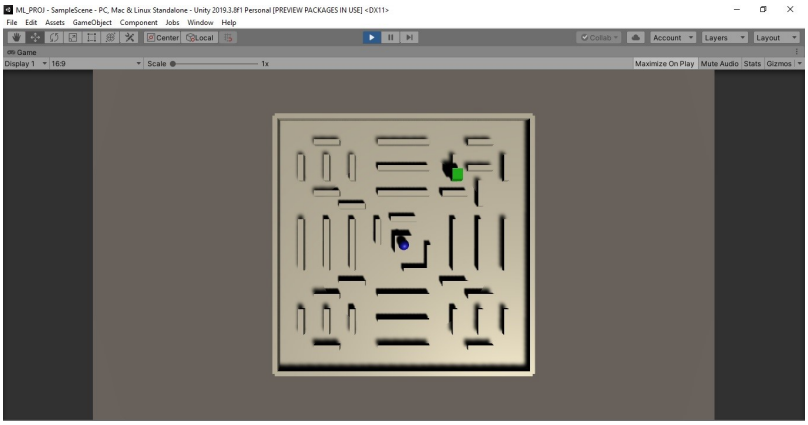

Figure 6: Maze 2

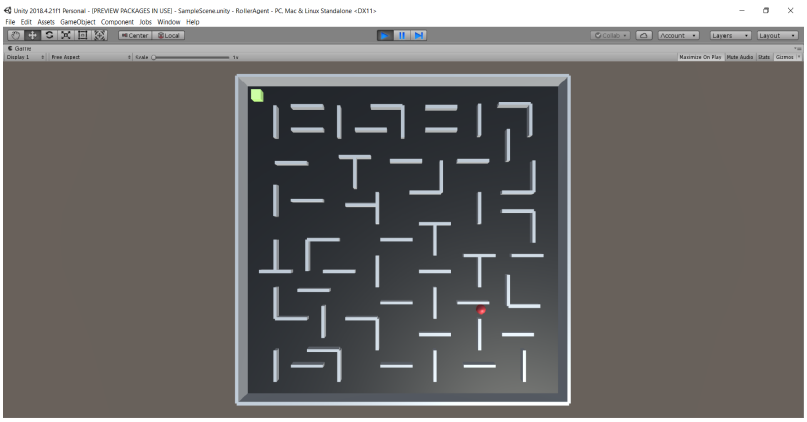

Figure 7: Maze 3

5 Conclusion and future work

In this project work, we successfully solved the maze problem using PPO along with curriculum learning. Curriculum learning along with Rewarding shaping allows the algorithm to converge faster to an optimal policy. Curriculum learning is also found to be viable in case of a dynamic environment for faster convergence. The input features for the neural network are the most important deciding factor that determines the performance of the model, for example in our case the ray perception sensor gathers the right observations(neural network input features) to attain the optimal policy. In this way experimenting with different input features for the neural network and Unity features helped in optimizing the result.

As part of our future work we would like to work on dynamic and com-

plex environment. For even more better understanding we would like to do a comparison with other policy gradient based algorithms.

References

[1] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. (2017). Proximal Policy Optimization Algorithms. ArXiv, abs/1707.06347.

[2] Arthur, J., Vincent-Pierre, B., Ervin, T., Andrew, C., Jonathan, H., Chris, E., Chris, G., Yuan, G., Hunter, H., Marwan, M., Danny, L. (2018). Unity: A General Platform for Intelligent Agents. ArXiv, arXiv:1809.02627.

[3] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. In Proceedings of the 12th International Conference on Neural Information Processing Systems (NIPS'99). MIT Press, Cambridge, MA, USA, 1057–1063.

[4] Laud, Adam Daniel.(2004-05).Theory and Application of Reward Shaping in Reinforcement Learning, http://hdl.handle.net/2142/10797

[5] B. Qin, Y. Gao and Y. Bai, "Sim-to-real: Six-legged Robot Control with Deep Reinforcement Learning and Curriculum Learning," 2019 4th International Conference on Robotics and Automation Engineering (ICRAE), Singapore, Singapore, 2019, pp. 1-5, doi: 10.1109/ICRAE48301.2019.9043822.